Chapter 17

Answers to exercises

We give in this chapter one possible solution for each exercise contained in this document. Exercises are referred to by their number and the page where they have been proposed: for example, "2.1, p. 15" refers to the first exercise in chapter 2; this exercise is located on page 15.

3.1, p. 19

The following (anonymous) functions have the required types:

1.
```
#function f -> (f 2)+1;;
- : (int -> int) -> int = <fun>
```

2.
```
#function m -> (function n -> n+m+1);;
- : int -> int -> int = <fun>
```

3.
```
#(function f -> (function m -> f(m+1) / 2));;
- : (int -> int) -> int -> int = <fun>
```

3.2, p. 19

We must first rename `y` to `z`, obtaining:

```
(function x -> (function z -> x+z))
```

and finally:

```
(function y -> (function z -> y+z))
```

Without renaming, we would have obtained:

```
(function y -> (function y -> y+y))
```

which does not denote the same function.

3.3, p. 19

We write successively the reduction steps for each expressions, and then we use Caml in order to check the result.

- ```
  let x=1+2 in ((function y -> y+x) x);;
  (function y -> y+(1+2)) (1+2);;
  (function y -> y+(1+2)) 3;;
  3+(1+2);;
  3+3;;
  6;;
  ```
  Caml says:

```
#let x=1+2 in ((function y -> y+x) x);;
- : int = 6
```

- ```
  let x=1+2 in ((function x -> x+x) x);;
  (function x -> x+x) (1+2);;
  3+3;;
  6;;
  ```
  Caml says:

```
#let x=1+2 in ((function x -> x+x) x);;
- : int = 6
```

- ```
  let f1 = function f2 -> (function x -> f2 x)
  in let g = function x -> x+1
     in f1 g 2;;
  let g = function x -> x+1
  in function f2 -> (function x -> f2 x) g 2;;
  (function f2 -> (function x -> f2 x)) (function x -> x+1) 2;;
  (function x -> (function x -> x+1) x)  2;;
  (function x -> x+1) 2;;
  2+1;;
  3;;
  ```
  Caml says:

```
#let f1 = function f2 -> (function x -> f2 x)
#in let g = function x -> x+1
#   in f1 g 2;;
- : int = 3
```

4.1, p. 31

To compute the surface area of a rectangle and the volume of a sphere:
```

```
#let surface_rect len wid = len * wid;;
surface_rect : int -> int -> int = <fun>
```

```
#let pi = 4.0 *. atan 1.0;;
pi : float = 3.14159265359
```

```
#let volume_sphere r = 4.0 /. 3.0 *. pi *. (power r 3.);;
volume_sphere : float -> float = <fun>
```

4.2, p. 31

In a call-by-value language without conditional construct (and without sum types), all programs involving a recursive definition never terminate.

4.3, p. 31

```
#let rec factorial n = if n=1 then 1 else n*(factorial(n-1));;
factorial : int -> int = <fun>
```

```
#factorial 5;;
- : int = 120
```

```
#let tail_recursive_factorial n =
#  let rec fact n m = if n=1 then m else fact (n-1) (n*m)
#  in fact n 1;;
tail_recursive_factorial : int -> int = <fun>
```

```
#tail_recursive_factorial 5;;
- : int = 120
```

4.4, p. 31

```
#let rec fibonacci n =
#  if n=1 then 1
#         else if n=2 then 1
#                     else fibonacci(n-1) + fibonacci(n-2);;
fibonacci : int -> int = <fun>
```

```
#fibonacci 20;;
- : int = 6765
```

4.5, p. 32

```
#let compose f g = function x -> f (g (x));;
compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

```
#let curry f = function x -> (function y -> f(x,y));;
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

```
#let uncurry f = function (x,y) -> f x y;;
uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>

#uncurry compose;;
- : ('_a -> '_b) * ('_c -> '_a) -> '_c -> '_b = <fun>

#compose curry uncurry;;
- : ('_a -> '_b -> '_c) -> '_a -> '_b -> '_c = <fun>

#compose uncurry curry;;
- : ('_a * '_b -> '_c) -> '_a * '_b -> '_c = <fun>
```

5.1, p. 36

```
#let rec combine =
#  function [],[] -> []
#         | (x1::l1),(x2::l2) -> (x1,x2)::(combine(l1,l2))
#         | _ -> raise (Failure "combine: lists of different length");;
combine : 'a list * 'b list -> ('a * 'b) list = <fun>

#combine ([1;2;3],["a";"b";"c"]);;
- : (int * string) list = [1, "a"; 2, "b"; 3, "c"]

#combine ([1;2;3],["a";"b"]);;
Uncaught exception: Failure "combine: lists of different length"
```

5.2, p. 36

```
#let rec sublists =
#    function [] -> [[]]
#           | x::l -> let sl = sublists l
#                     in sl @ (map (fun l -> x::l) sl);;
sublists : 'a list -> 'a list list = <fun>

#sublists [];;
- : '_a list list = [[]]

#sublists [1;2;3];;
- : int list list = [[]; [3]; [2]; [2; 3]; [1]; [1; 3]; [1; 2]; [1; 2; 3]]

#sublists ["a"];;
- : string list list = [[]; ["a"]]
```

6.1, p. 46

```
#type ('a,'b) btree = Leaf of 'b
#                   | Btree of ('a,'b) node
#and ('a,'b) node = {Op:'a;
```

```
#                    Son1: ('a,'b) btree;
#                    Son2: ('a,'b) btree};;
Type btree defined.
Type node defined.
```

```
#let rec nodes_and_leaves =
#    function Leaf x -> ([],[x])
#           | Btree {Op=x; Son1=s1; Son2=s2} ->
#                let (nodes1,leaves1) = nodes_and_leaves s1
#                and (nodes2,leaves2) = nodes_and_leaves s2
#                in (x::nodes1@nodes2, leaves1@leaves2);;
nodes_and_leaves : ('a, 'b) btree -> 'a list * 'b list = <fun>
```

```
#nodes_and_leaves (Btree {Op="+"; Son1=Leaf 1; Son2=Leaf 2});;
- : string list * int list = ["+"], [1; 2]
```

6.2, p. 46

```
#let rec map_btree f g = function
#      Leaf x -> Leaf (f x)
#    | Btree {Op=op; Son1=s1; Son2=s2}
#               -> Btree {Op=g op; Son1=map_btree f g s1;
#                                  Son2=map_btree f g s2};;
map_btree : ('a -> 'b) -> ('c -> 'd) -> ('c, 'a) btree -> ('d, 'b) btree =
 <fun>
```

6.3, p. 46

We need to give a functional interpretation to `btree` data constructors. We use `f` (resp. `g`) to denote the function associated to the `Leaf` (resp. `Btree`) data constructor, obtaining the following Caml definition:

```
#let rec btree_it f g = function
#      Leaf x -> f x
#    | Btree{Op=op; Son1=s1; Son2=s2}
#              -> g op (btree_it f g s1) (btree_it f g s2)
#;;
btree_it : ('a -> 'b) -> ('c -> 'b -> 'b -> 'b) -> ('c, 'a) btree -> 'b =
 <fun>
```

```
#btree_it (function x -> x)
#         (function "+" -> prefix +
#                 | _ -> raise (Failure "Unknown op"))
#         (Btree {Op="+"; Son1=Leaf 1; Son2=Leaf 2});;
- : int = 3
```

7.1, p. 54

```
#type ('a,'b) lisp_cons = {mutable Car:'a; mutable Cdr:'b};;
Type lisp_cons defined.

#let car p = p.Car
#and cdr p = p.Cdr
#and rplaca p v = p.Car <- v
#and rplacd p v = p.Cdr <- v;;
car : ('a, 'b) lisp_cons -> 'a = <fun>
cdr : ('a, 'b) lisp_cons -> 'b = <fun>
rplaca : ('a, 'b) lisp_cons -> 'a -> unit = <fun>
rplacd : ('a, 'b) lisp_cons -> 'b -> unit = <fun>

#let p = {Car=1; Cdr=true};;
p : (int, bool) lisp_cons = {Car=1; Cdr=true}

#rplaca p 2;;
- : unit = ()

#p;;
- : (int, bool) lisp_cons = {Car=2; Cdr=true}
```

7.2, p. 54

```
#let stamp_counter = ref 0;;
stamp_counter : int ref = ref 0

#let stamp () =
#  stamp_counter := 1 + !stamp_counter; !stamp_counter;;
stamp : unit -> int = <fun>

#stamp();;
- : int = 1

#stamp();;
- : int = 2
```

7.3, p. 54

```
#let exchange t i j =
#  let v = t.(i) in vect_assign t i t.(j); vect_assign t j v
#;;
exchange : 'a vect -> int -> int -> unit = <fun>

#let quick_sort t =
#  let rec quick lo hi =
#    if lo < hi
#    then begin
```

```
#           let i = ref lo
#           and j = ref hi
#           and p = t.(hi) in
#           while !i < !j
#            do
#             while !i < hi & t.(!i) <=. p do incr i done;
#             while !j > lo & p <=. t.(!j) do decr j done;
#             if !i < !j then exchange t !i !j
#            done;
#           exchange t hi !i;
#           quick lo (!i - 1);
#           quick (!i + 1) hi
#          end
#      else ()
#   in quick 0 (vect_length t - 1)
#;;
quick_sort : float vect -> unit = <fun>

#let a = [| 2.0; 1.5; 4.0; 0.0; 10.0; 1.0 |];;
a : float vect = [|2.0; 1.5; 4.0; 0.0; 10.0; 1.0|]

#quick_sort a;;
- : unit = ()

#a;;
- : float vect = [|0.0; 1.0; 1.5; 2.0; 4.0; 10.0|]
```

8.1, p. 58

```
#let rec find_succeed f = function
#       [] -> raise (Failure "find_succeed")
#     | x::l -> try f x; x with _ -> find_succeed f l
#;;
find_succeed : ('a -> 'b) -> 'a list -> 'a = <fun>

#let hd = function [] -> raise (Failure "empty") | x::l -> x;;
hd : 'a list -> 'a = <fun>

#find_succeed hd [[];[];[1;2];[3;4]];;
- : int list = [1; 2]
```

8.2, p. 58

```
#let rec map_succeed f = function
#       [] -> []
#     | h::t -> try (f h)::(map_succeed f t)
#                with _ ->  map_succeed f t;;
```

```
map_succeed : ('a -> 'b) -> 'a list -> 'b list = <fun>

#map_succeed hd [[];[1];[2;3];[4;5;6]];;
- : int list = [1; 2; 4]
```

9.1, p. 63

The first function (copy) that we define assumes that its arguments are respectively the input and the output channel. They are assumed to be already opened.

```
#let copy inch outch =
#  (* inch and outch are supposed to be opened channels *)
#  try (* actual copying *)
#      while true
#      do output_char outch (input_char inch)
#      done
#   with End_of_file -> (* Normal termination *)
#             raise End_of_file
#      | sys__Sys_error msg -> (* Abnormal termination *)
#             prerr_endline msg;
#             raise (Failure "cp")
#      | _ -> (* Unknow exception, maybe interruption? *)
#             prerr_endline "Unknown error while copying";
#             raise (Failure "cp")
#;;
copy : in_channel -> out_channel -> unit = <fun>
```

The next function opens channels connected to its filename arguments, and calls copy on these channels. The advantage of dividing the code into two functions is that copy performs the actual work, and can be reused in different applications, while the role of cp is more "administrative" in the sense that it does nothing but opening and closing channels and printing possible error messages.

```
#let cp f1 f2 =
#  (* Opening channels, f1 first, then f2 *)
#  let inch =
#      try open_in f1
#      with sys__Sys_error msg ->
#                prerr_endline (f1^": "^msg);
#                raise (Failure "cp")
#         | _ -> prerr_endline ("Unknown exception while opening "^f1);
#                raise (Failure "cp")
#  in
#  let outch =
#      try open_out f2
#      with sys__Sys_error msg ->
```

```
#                close_in inch;
#                prerr_endline (f2^": "^msg);
#                raise (Failure "cp")
#          | _ -> close_in inch;
#                prerr_endline ("Unknown exception while opening "^f2);
#                raise (Failure "cp")
#  in (* Copying and then closing *)
#     try copy inch outch
#     with End_of_file -> close_in inch; close_out outch
#                        (* close_out flushes *)
#        | exc -> close_in inch; close_out outch; raise exc
#;;
cp : string -> string -> unit = <fun>
```

Let us try cp:

```
#cp "/etc/passwd" "/tmp/foo";;
- : unit = ()
```

```
#cp "/tmp/foo" "/foo";;
/foo: /foo: Permission denied
Uncaught exception: Failure "cp"
```

The last example failed because a regular user is not allowed to write at the root of the file system.

9.2, p. 63

As in the previous exercise, the function count performs the actual counting. It works on an input channel and returns a pair of integers.

```
#let count inch =
#    let chars = ref 0
#    and lines = ref 0 in
#    try
#      while true do
#        let c = input_char inch in
#          chars := !chars + 1;
#          if c = `\n` then lines := !lines + 1 else ()
#      done;
#      (!chars, !lines)
#    with End_of_file -> (!chars, !lines)
#;;
count : in_channel -> int * int = <fun>
```

The function wc opens a channel on its filename argument, calls count and prints the result.

```
#let wc f =
#    let inch =
```

```
#       try open_in f
#       with sys__Sys_error msg ->
#                prerr_endline (f^": "^msg);
#                raise (Failure "wc")
#         | _ -> prerr_endline ("Unknown exception while opening "^f);
#                raise (Failure "wc")
#  in let (chars,lines) = count inch
#     in   print_int chars;
#          print_string " characters, ";
#          print_int lines;
#          print_string " lines.\n"
#;;
wc : string -> unit = <fun>
```

Counting `/etc/passwd` gives:

```
#wc "/etc/passwd";;
32474 characters, 392 lines.
- : unit = ()
```

10.1, p. 76

Let us recall the definitions of the type `token` and of the lexical analyzer:

```
#type token =
#  PLUS | MINUS | TIMES | DIV | LPAR | RPAR
#| INT of int;;
Type token defined.
```

```
#(* Spaces *)
#let rec spaces = function
#  [< '` `|`\t`|`\n`; spaces _ >] -> ()
#| [< >] -> ();;
spaces : char stream -> unit = <fun>
```

```
#(* Integers *)
#let int_of_digit = function
#  `0`..`9` as c -> (int_of_char c) - (int_of_char `0`)
#| _ -> raise (Failure "not a digit");;
int_of_digit : char -> int = <fun>
```

```
#let rec integer n = function
#  [< ' `0`..`9` as c; (integer (10*n + int_of_digit c)) r >] -> r
#| [< >] -> n;;
integer : int -> char stream -> int = <fun>
```

```
#(* The lexical analyzer *)
#let rec lexer s = match s with
```

```
#  [< '`(`; spaces _ >] -> [< 'LPAR; lexer s >]
#| [< '`)`; spaces _ >] -> [< 'RPAR; lexer s >]
#| [< '`+`; spaces _ >] -> [< 'PLUS; lexer s >]
#| [< '`-`; spaces _ >] -> [< 'MINUS; lexer s >]
#| [< '`*`; spaces _ >] -> [< 'TIMES; lexer s >]
#| [< '`/`; spaces _ >] -> [< 'DIV; lexer s >]
#| [< '`0`..`9` as c; (integer (int_of_digit c)) n; spaces _ >]
#                                -> [< 'INT n; lexer s >];;
lexer : char stream -> token stream = <fun>
```

The parser has the same shape as the grammar:

```
#let rec expr = function
#  [< 'INT n >] -> n
#| [< 'PLUS; expr e1; expr e2 >] -> e1+e2
#| [< 'MINUS; expr e1; expr e2 >] -> e1-e2
#| [< 'TIMES; expr e1; expr e2 >] -> e1*e2
#| [< 'DIV; expr e1; expr e2 >] -> e1/e2;;
expr : token stream -> int = <fun>

#expr (lexer (stream_of_string "1"));;
- : int = 1

#expr (lexer (stream_of_string "+ 1 * 2 4"));;
- : int = 9
```

10.2, p. 76

The only new function that we need is a function taking as argument a character stream, and returning the first identifier of that stream. It could be written as:

```
#let ident_buf = make_string 8 ` `;;
ident_buf : string = "        "

#let rec ident len = function
#  [< ' `a`..`z`|`A`..`Z` as c;
#     (if len >= 8 then ident len
#      else begin
#            set_nth_char ident_buf len c;
#            ident (succ len)
#           end) s >] -> s
#| [< >] -> sub_string ident_buf 0 len;;
ident : int -> char stream -> string = <fun>
```

The lexical analyzer will first try to recognize an alphabetic character *c*, then put *c* at position 0 of `ident_buf`, and call `ident 1` on the rest of the character stream. Alphabetic characters encountered will be stored in the string buffer `ident_buf`, up to the 8th. Further alphabetic characters will be skipped. Finally, a substring of the buffer will be returned as result.

```
#let s = stream_of_string "toto 1";;
s : char stream = <abstr>

#ident 0 s;;
- : string = "toto"

#(* Let us see what remains in the stream *)
#match s with [< 'c >] -> c;;
- : char = ` `

#let s = stream_of_string "LongIdentifier ";;
s : char stream = <abstr>

#ident 0 s;;
- : string = "LongIden"

#match s with [< 'c >] -> c;;
- : char = ` `
```

The definitions of the new `token` type and of the lexical analyzer is trivial, and we shall omit them. A slightly more complex lexical analyzer recognizing identifiers (lowercase only) is given in section 12.2.1 in this part.

11.1, p. 81

```
(* main.ml *)
let chars = counter__new 0;;
let lines = counter__new 0;;

let count_file filename =
  let in_chan = open_in filename in
    try
      while true do
        let c = input_char in_chan in
          counter__incr chars;
          if c = `\n` then counter__incr lines
      done
    with End_of_file ->
      close_in in_chan
;;

for i = 1 to vect_length sys__command_line - 1 do
  count_file sys__command_line.(i)
done;
print_int (counter__read chars);
print_string " characters, ";
print_int (counter__read lines);
print_string " lines.\n";
exit 0;;
```